CONTEXTUAL STATIC ANALYSIS

Value Injection in Conjunction with Static Analysis
Capabilities Brief

(Sample Chapters and Table of Contents)

Table of Contents

1.0 Executive Summary

The complex integration of software systems that run mission critical and safety critical infrastructure multiplies the opportunities for and severity of software flaws shipped with those systems. The expense of post-deployment remediation, though often 100 times more costly than catching problems prior to production, is also sometimes measured in human lives.

Highly automated tools are emerging for analyzing source code in its original, pre-compiled state (static source code analysis). This allows organizations to avoid expensive late stage testing (such as integration testing) to discover flaws and vulnerabilities. Few of these tools provide a system approach that allows interaction during test, forcing auditors to standby while analysis proceeds. A major limitation of static source code analysis is the inability to resolve run-time values, that is values defined during program execution. Without a known set of values to use it is impossible for these systems to return accurate, meaningful results pertaining to dependent lines of execution which rely on such values. The resulting analysis identifies only “indeterminate” errors (and sometimes none at all), giving auditors little to go on and relegating further investigation to the same intensive manual code reviews that automated analysis was intended to avoid.

The unique approach taken by Toolbuilders Laboratories provides a system that prompts auditors to inject values during static analysis, then gives them the ability to visually trace all dependent execution through the entire code project. The result is a sophisticated code improvement environment focused on dramatically increasing the auditor’s code comprehension and ultimately the ability to eradicate software problems on a system wide scale.

1.1 Detailed Problem Profile

Mission critical applications used in sectors such as national defense, intelligence, utilities, financial institutions and telecommunications industries, comprise billions of lines of source code. These large amalgamations require painstakingly meticulous development, quality assurance (QA) and maintenance processes to guard against flaws that can lead to unexpected behavior or make them vulnerable to attacks. With a manual repair cost of $1.50 to $2.50 per line of code, the annual cost of minimizing security vulnerabilities or defects is staggering. With current methods, software analysis and repairs are time-consuming and alarmingly incomplete.

Studies have shown that the earlier a defect is found in the development cycle the cheaper it is to repair. Some studies suggest the ratio is greater than 10 to 1. The emphasis of our business is to maximize the return on investment for our customers by delivering products that streamline the development process as it relates to the early detection and correction of errors.

Manually reviewing thousands of lines of source code is tedious ,prone to human error, and lacks sufficient code coverage. Automated and semi-automated systems that can analyze the original source code prior to compilation (static source code analysis) have started to appear as a means of alleviating the problem; these systems mostly focus on Quality Assurance phases of the development cycle, omitting code authors from an active role in early error detection. Though it has many benefits over other methods, a major limitation of static source code analysis is the inability to resolve all possible run-time values. With current static analysis tools this typically leads to a large number of unresolved variables and therefore a large number of warnings that are either false positives or indeterminate in nature. This significantly reduces the effectiveness of the analysis because developers and QA personnel cannot readily determine what is important and what is not without again reverting to intensive manual review.

Traditional approaches to solve the problem have been to use abstract interpretation for static analysis or manual runtime testing, relegating such activities to specialized QA personnel and thereby missing the opportunity to deliver investigative auditing tools into the hands of developers while code is being written . Abstract interpretation has limited usefulness because of resource utilization requirements and poor performance. Manual testing is a very difficult activity to perform to check every possible path under every possible scenario.

1.2 The Value Injection Backtrace Approach

The approach taken by Toolbuilders Laboratories provides a bridge between the two extremes of detailed manual testing and abstract interpretation, and to give code authors development features that naturalize code improvement activities into their standard practices It suggests that significant value can be added to the development process by providing “value injection” capabilities within a static analysis tool where a typical unresolved variable situation would exist. Value injection means providing an opportunity during static analysis to define a finite value or set of values for a variable whenever such variables would otherwise cause ambiguous analysis results. By capturing the values defined for variables in this manner, and following the variables through all transformations along multiple call paths, value injection promises performance advantages because it only requires those items to be injected at a specific point in the call graph, and restricts the operation just to those items being analyzed. This can lead to a significant analysis performance enhancement over abstract interpretation and move error correction to the earliest point possible in the development process

The usability of this approach is further enhanced by giving the developer an intuitive visualization of the context of the variable requiring value injection. Once the context is understood, the proposed system can propagate that value throughout the call graph as necessary to resolve other variables. The measurable value is the resulting process improvement from developers that can now aggressively participate in the discovery and correction of errors while accelerating their delivery of production software.

Armed with this approach, a developer or tester can better simulate the run-time application environment and garner the benefits of total path coverage because the value will be injected into all possible paths of the application.

1.3 Management Approach

The management controls necessary to prevent “rabbit trail” research is important in an effort such as this because focusing upon the single goal amongst many reasonable possibilities can be difficult. Using Extreme Programming (XP) as a management and development methodology focuses efforts and reduces the challenge.

XP development methodologies dictate that interaction between clients and staff is critical as the information exchange that occurs is the lifeblood of a successful development effort. This communication takes the form of status reports, up-to-date XP stories and relatively frequent in-process releases.

2.0 Technical Approach

The ambiguous analysis results that are often produced by static analysis can be mitigated through value injection. Value injection will provide an opportunity during static analysis to define a finite value for a variable to produce tangible results. When saved for this purpose and applied to all variable transformations along multiple call paths, performance advantages are easily recognized. These performance advantages occur because values must only be injected at a specific point in the call graph, and the operation is restricted to just those items being analyzed.

2.1 Current Approaches

Contemporary techniques require a heavy reliance on tools and methods that require intensive human effort through meticulous, error-prone review and intervention processes in order to correct the kinds of defects better captured through value injection.

2.1.1 Simple manual testing

Manual testing is the basic testing philosophy that has been followed in all organizations for years. This approach assumes that the testers understand, in enough detail, the application to effectively test it thoroughly. As evidenced by almost any current publication, that is a flawed approach.

Studies indicate that the average human mind can remember seven things repeatedly and accurately. Since most programs are considerably more complex than that, human testers are at an immediate disadvantage and significantly limiting the effectiveness of the testing being accomplished.

The very simple source code sample below illustrates the problem.

globals.h

```
char msg_buf[ 1024 ];
```

build_msg.c

```
#include “globals.h”

char msg_buf[ 512 ];

void build_message( char *theMessage )
{
      sprintf( msg_buf, “the msg: %s\n”, theMessage );
}
```

app_msg.c

```
void extract_and_build_app_msg( void )
{
      char msg[ 1024 ];
      extract_msg( msg );
      build_message( msg );
}
```

Assuming that the manual tester is an effective C coder, they would understand that scoping rules require that the variable “msg_buf” in build_msg.c is used and that a potential buffer overflow condition

has occurred. This is because the parameter “msg” has transitioned into “theMessage” at the call interface and “msg_buf” is a static array value declared after the included file.

In this simplistic example where all of the code is shown together it is easy to see what to do. Now consider the case where there are several thousand lines of code between the declaration of the buffer and the use of it. This is quite common in auto-generated code such as MATLAB where blocks of code are glued together to form methods that relate to a diagram. This can become an exceedingly difficult problem to undertake very quickly.

Another much more difficult manual test scenario can be imagined from this sample code. The function “build_message” is used as a utility function across multiple modules within the application. Now instead of just tracking down the single instance of the parameter value the tester must verify all of the potential values of the parameter. This can be a Herculean task for a human to try and accomplish effectively.

2.1.2 Manual testing with path coverage tool

As noted in section 3.1.1 manual testing alone is very difficult to rely on when trying to prove out an application. Path coverage tools are used to help a tester become more confident that all possible paths through the code have been reviewed and verified.

While certainly helpful in determining the paths the application might take through the code, it is still the effort of the human that decided whether a particular path is feasible and adequately verified. On even moderately sized projects that remains a Herculean task.

2.1.3 Abstract Interpretation

Abstract Interpretation is a theory for formally constructing conservative approximations of the semantics of programming languages. In practice, it is used for constructing semantics-based analysis algorithms for the automatic, static and conservative determination of dynamic properties of infinite-state programs. Such properties of the run-time behavior of programs are useful for debugging (e.g. type inference), code optimization (e.g. run time tests elimination), program transformation (e.g. partial evaluation, parallelization), and even program correctness proofs (e.g. termination proof).

According to the classical framework put forth by Cousot and Cousot, an abstract interpretation is defined as a non-standard (approximated) program semantics obtained from the standard (or concrete) one by replacing the actual (concrete) domain of computation and its basic (concrete) semantic operations with, respectively, an *abstract domain* and corresponding *abstract semantic operations*.

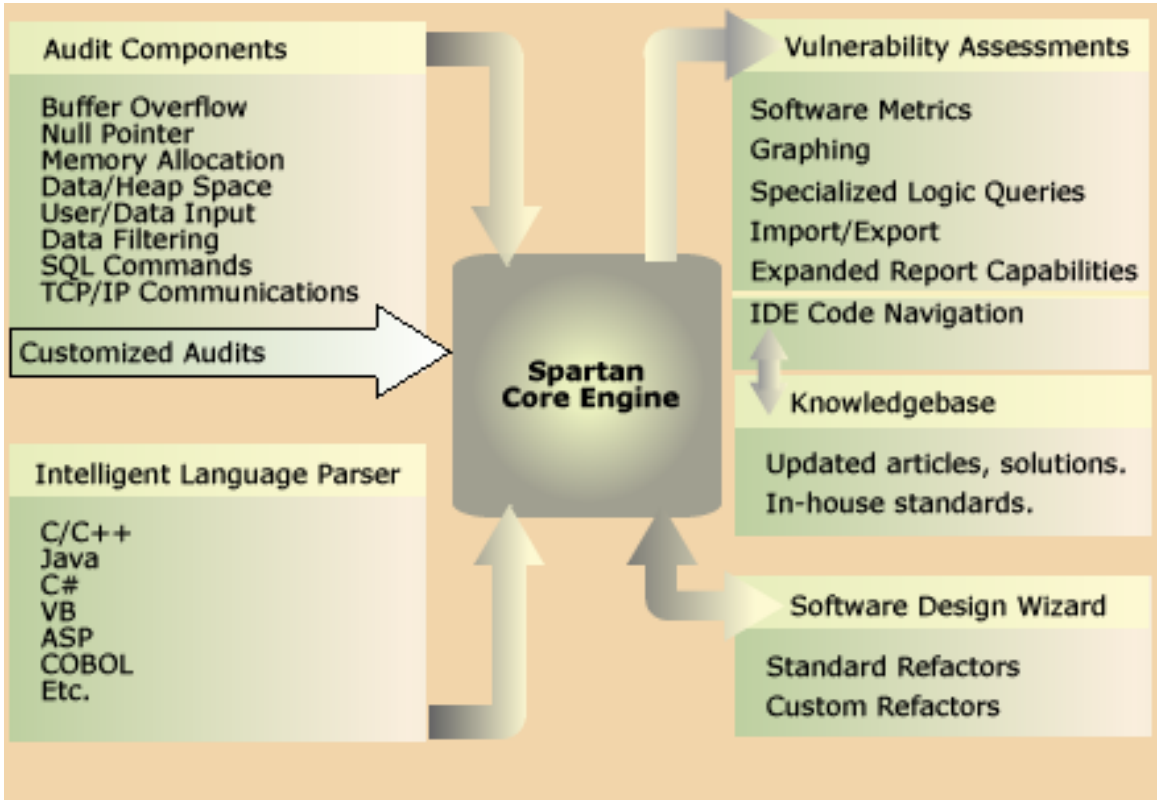
Because abstract interpretation is a mathematically intensive approach, very few, if any, shortcuts can be taken. This leads to extremely high resource needs and relatively poor performance overall. A leading abstract interpretation product vendor has a performance benchmark on their web site of 50,000 lines of code analyzed in two weeks. That benchmark is not particularly appealing when required to test a project with a million lines of source code. A typical compiler benchmark for a similar sized project would be no more than a few minutes. Sempre does an analysis of a somewhat larger project, about 80,000 lines of code, in approximately four minutes.

2.2 Proposed Approach

Using backtrace visualization algorithms to aggressively map the complex call graph of an application, a method will be devised for the efficient delivery of value injection capability providing faster, more accurate static analysis of variables that would otherwise return indeterminate analysis errors. This approach will increase analysis accuracy by using the injected values throughout the call graph, as appropriate, and following identified variables through all transformations.

2.2.1 Architecture

The current Sempre architectural implementation is shown below.



The architecture of the proposed solution is to build a substantially similar, but minimized, version of Sempre as a prototyping tool. Once that is accomplished, extend the GUI to support editing of values within the backtrace. That then will cause a re-computation of the values from the point of data entry to take into account the injected value.

After the prototyped proof of concept is proved out, additional work will be done to implement the resulting solution seamlessly into Sempre or other technologies.

2.2.2 Backtrace Generation

Backtrace generation is a critical step along the path from parsing to visualization. It builds the call and context graph from which all values and sizes are derived. The fundamental work being done by backtrace generation is connecting audited items all the way back to the declaration. This can be accomplished through variable transformations such as simple assignments or more complex assignments such as function calls. The code below illustrates the differences.

```
void msg()
```

```
{
      int b;
      int a = get_length( “some string” );
      if ( a < 100 )
            b = a;
}

int get_length( char *value )
{
      return strlen( value );
}
```

While the example above is rather simplistic and unlikely to be found in production code, it does point out the complexity in tracing values to their ultimate size and value. In this particular case for the value of variable “a” to be accurately determined means that an analysis would have to evaluate the context of the function “get_length” to see what is actually returned.

The problem becomes acute when a variable is passed by reference instead of by value as shown below.

```
void msg()
{
      char str_buf[ 100 ] = “some string”;
      int b;
      int a = get_length( str_buf );
      if ( a < 100 )
            b = a;
}

int get_length( char *value )
{
      int len = strlen( value );
      if ( len < 100 )
      {
            strcpy( value, “string not long enough” );
            len = strlen( value );
      }

      return len;
}
```

In the example above, not only is it necessary to track the length returned, but also the underlying value of the variable changed during the call. It is also possible to inject a value into the variable “str_buf” to allow for checking both the less than 100 condition and the greater than 100 condition. This simple example shows the power of value injection technology.

2.2.3 Visualization

Backtrace visualization provides function and variable tracing through large volumes of code for the rapid determination of their value and size. This gives developers assisted manual audit capability by providing clear, precise views of complex application relationships. This behavior is a dramatic code visualization enhancement feature not found in current analysis tools that may look at limited aspects of code context.

Research suggests the optimal delivery of visualization features will include the concept of Visual Backtrace, possibly to be presented in a tree structure that allows concise code navigation. Toolbuilders

Laboratories has already completed significant research into this aspect of analysis, but more is needed to determine the best delivery of value injection capability using aspects of Visual Backtracing capabilities. Experience to date suggests that unlike call trees that are represented as a series of nodes and connecting lines (spider web), backtrace visualization is better presented as a tree structure within interactive tabular list. Clarity is achieved by having each entry linked to actual code to provide detailed descriptions of the nodes. A role that is linked to the variable or function as assignment, declaration, parameter, or usage also helps with clarity. More research will determine how best to develop backtrace visualization capabilities and displays optimized for value injection delivery.

Exhaustive parameter tracing is perhaps the most powerful and promising element of backtrace visualization algorithms that incorporate value injection capabilities. The value of the parameter must be traced along the backtrace path, from selection to declaration, to determine if its value or size changes anywhere along the path. Without runtime information it is crucial to follow every transformation of a parameter, and not just the value that is passed. These transformations include pointer arithmetic, struct membership, and expression evaluation.

Experience suggests that backtrace visualization should not only trace the individual named parameters of a function, but also all constructs that impact the value of the named parameter. This results in the most complete information possible without runtime information. As it relates to value injection, the relevance of backtrace visualization is expected to be high.

2.2.4 Value Injection and Persistence

The true power of value injection can be shown in the following example. While this is again a rather trivial example, there is no reason that the concept cannot be extended to far more complex production situations. With the aid of an accurate backtrace, and a powerful visualization, the complexity almost becomes irrelevant.

```
void msg()
{
      char str_buf[ 100 ] = “some string”;
      int b;
      int a = get_length( str_buf );
      if ( a < 100 )
            b = a;

      print_value( str_buf );
}

int get_length( char *value )
{
      int len = strlen( value );
      if ( len < 100 )
      {
            strcpy( value, “string not long enough” );
            len = strlen( value );
      }

      return len;
}

void print_value( char *value )
{
      printf( “%s\n”, value );
}
```

Please note that the variable “str_buf” might be modified prior to the call to the function “print_value”. Using value injection would provide a easy mechanism to test both paths through the code and assist in proving it out.

2.3 Conclusion

Studies have shown that the earlier a defect is found in the development cycle the cheaper it is to repair. Some studies suggest the ratio is greater than 10 to 1. The emphasis of our business is to maximize the return on investment for our customers. The expertise of our staff in conjunction with the efficiency of our products leads to an effective delivery mechanism.

Using our proven technology as the basis for solving some of the industry’s most daunting application development problems, we intend to provide an analysis solution that provides significant advantage to Verification &Validation processes and the proactive deployment of secure, stable software.

With the added capability of value injection that vision can be made manifest for developers and auditors today. As has been demonstrated, backtrace generation requires definitive skill that is possessed by the developers at Toolbuilders Laboratories; extending it to allow for value injection is a goal that is within reach.